Mining Relationships between the Participants of Architectural Patterns

Kamal, Ahmad Waqas; Avgeriou, Paris

Published in:
EPRINTS-BOOK-TITLE

Mining Relationships between the Participants of Architectural Patterns

Ahmad Waqas Kamal and Paris Avgeriou

Department of Mathematics and Computing Science,
University of Groningen, The Netherlands
a.w.kamal@rug.nl, paris@cs.rug.nl

Abstract. Architectural patterns are often combined with other, relevant architectural patterns during software architecture design. However, combining patterns effectively remains a challenging task: first because the integration of any two architectural patterns can take several forms; second because existing pattern languages only mention generic pattern-to-pattern relationships and do not go into the details of their combination. In this paper, we propose to address this problem by discovering and defining a handful of recurring pattern relationships at the level of the participants of patterns. We have studied 32 industrial case studies and mined a number of relationships between participants of different patterns. We present a few of these relationships and outline some examples of their appearance.

Keywords: Architectural Patterns, Pattern relationships, Pattern Languages.

1 Introduction

Over the last decade, architectural patterns have increasingly become an integral part of software architecture design practices [1]. Architectural patterns are seldom applied in isolation within a software architecture: individual architectural patterns can only solve specific parts of the design problem; it takes a combination of patterns to cover all the requirements. For instance, the Client-Server and Broker patterns are often used in combination to design distributed systems architectures [2]. Architectural patterns are characterized of intrinsic relations to other patterns, giving them the potential to solve larger design problems [3].

The integration of two or more patterns during software architecture design remains a challenging task. More precisely, we identify the following two challenges:

- Most of the pattern languages described in the literature document relationships among patterns at the conceptual level [4]. However, none of these pattern languages deals with the relationships among participants[1] of related patterns. In this sense, current pattern languages only offer guidance for the selection of related architectural patterns, or hints to design a particular kind of system; they do not provide support for integrating architectural patterns within software architecture

[1] The term pattern participants, frequently used in this paper, refers to the elements within the solution of patterns e.g. the Pipe and Filter are participants of the Pipes and Filters pattern.

M. Ali Babar and I. Gorton (Eds.): ECSA 2010, LNCS 6285, pp. 401–408, 2010.

design. Extensive design effort is required to precisely identify participants of related patterns that overlap, interact, or override related pattern participants in the resulting software architecture.
- Depending on the context of a system at hand, the combination of architectural patterns may entail variability, which is weakly addressed by existing pattern relationships approaches. For instance, to model interactive applications, the MVC[2] and Layers[2] pattern can be combined in several different forms like 3-tier layered architecture (where the presentation layer may consist of View and Control participants while the application logic layer owns the Model participant), which may vary for 2-tier or 4-tier software architectures.

To address the challenges described above, we propose to model the combination of patterns using a set of relationships between their participants. We have discovered these relationships by reviewing the patterns used in several industrial software architectures. We present a representative set of these relationships and exemplify them with instances from studied architectures.

The remainder of the paper is structured as follows: in Section 2, we describe the notion of pattern-to-pattern relationships and briefly outline the approach presented in this paper. In Section 3, we list the pattern participants relationships discovered during this work. Section 4 discusses the related work and Section 5 describes future work and concludes this study.

2 Relationships among Architectural Patterns

Architectural patterns are often combined with related patterns within software architectures. The value that individual patterns have, as solutions to design problems, is of course substantial, but their tremendous value comes when patterns are effectively combined within software architectures [1]: the combination of patterns is more than the sum of its parts. Unfortunately, individual patterns descriptions are not always explicit on 'how' to combine them with related architectural patterns. For instance, when reading the Reactor [4] pattern description, it is not clear how to apply Active Object [4] or Monitor Object [4]. In principle each participant within the solution of architectural patterns can be quite complex by itself, and often implemented using other patterns. It is therefore of paramount importance to express the intricate relationships between patterns, in order to effectively combine them within software architectures.

Pattern languages are thus far the most common and well-known form used by the software patterns community for defining relationships among architectural patterns. Pattern languages are not formal languages, although they document generic relationships among architectural patterns to address particular design problems [4]. For instance, the Model-View-Controller pattern has a 'change propagation' relationship with the Observer pattern as documented in [2]. Several pattern languages have been documented in the literature e.g. pattern languages for distributed computing [1], domain specific pattern languages [2], architectural views specific pattern language [3] etc.

2.1 The Proposed Approach

The underlying idea behind our approach is that architectural patterns can be effectively combined using a set of relationships between their participants. The relationships serve as a basis for identifying the participants of the patterns to be combined, that share responsibilities, overlap, or override each other. In order to come up with such relationships between participants, we have analyzed the architectural patterns used within several software architectures, and mined the relationships between their participants. The sources for eliciting pattern participants relationships were architecture design documents, architecture diagrams, design decisions, and case studies etc. The mined relationships must be recurring in several different examples in order to be considered as good design solutions. In the following section, we present a set of these relationships.

3 Mining Pattern-Participants Relationships for Modeling Patterns

The relationships presented in this section are based on the study of 32 industrial software architectures [5]. We provide, as an example, the complete documentation of a selected relationship. Due to space restrictions, a number of other relationships are also documented in an abbreviated form. We also provide a table to exemplify the discovered relationships by mapping pairs of patterns to the relationship between their participants.

3.1 Example of a Pattern Participant Relationship: absorbparticipant

In this sub-section, we present the detailed documentation of the absorbParticipant relationship while in the next sub-section we list several other relationships discovered during this study.

Definition: An *absorbParticipant* relationship defines how the participants of different patterns performing similar responsibilities are integrated in a single element. In an *absorbParticipant* relationship, certain participants of a pattern are absorbed by the participants of another pattern to avoid redundancy.

An example to describe the issue: Both the Reactor and Acceptor-Connector patterns introduce their own event handling solutions for using different services. The separate event handling structures in both patterns would be redundant if these patterns are applied in combination, e.g. the handler participant is present in both the Reactor and Acceptor-Connector patterns. In the Reactor's architectural structure, for each service an application offers, a separate event handler is introduced that processes different types of events from certain event sources. However, the Acceptor-Connector pattern can be suggested as an option to implement the Reactor's event handlers. This ensures that the Reactor pattern specifies the 'right' types of event handlers associated with the Acceptor-Connector pattern. In order to integrate the two patterns, the overlapping pattern participants either need to be merged or participants of one pattern be replaced by the other. However, removing a specific participant within a pattern may impact the solution specified by that pattern and may require new associations between the participants of both patterns, which is not a trivial work and require extensive design effort.

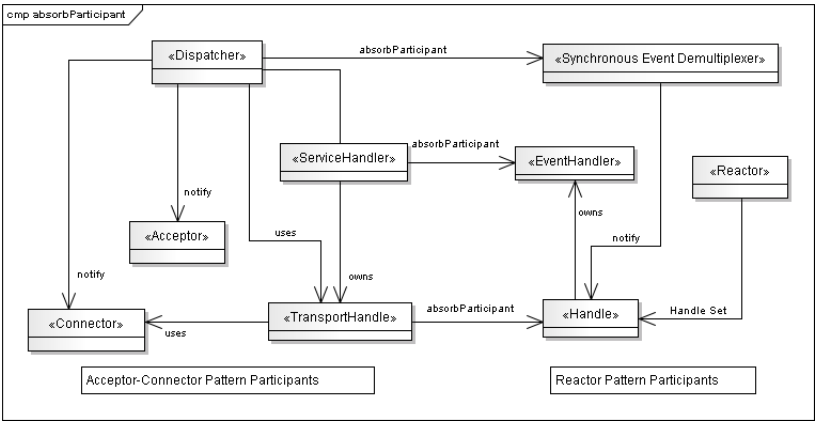

Fig. 1. The absorbParticipant relationship between Reactor and Acceptor-Connector

3.2 More Pattern Participants Relationships

Due to space restrictions, we will not go into detail for the rest of the pattern participants relationships we have elicited, but we will give a brief overview of these relationships.

mergeParticipant: The *mergeParticipant* relationship is used to combine one or more semantically different pattern participants into a single element within the target pattern. Such an integration retains the structural and semantic properties of individual participants into the target element. For instance, integrating the Active-Passive pattern with the Pipes and Filters pattern may result in certain filters passively processing the data. In essence, a passive filter in the Pipes and Filters chain performs the responsibilities of both a filter and a passive element. The *mergeParticipant* relationship is different from the *absorbParticipant* relationship where participants performing similar responsibilities are overridden (i.e. redundant participants are virtually removed in the resulting

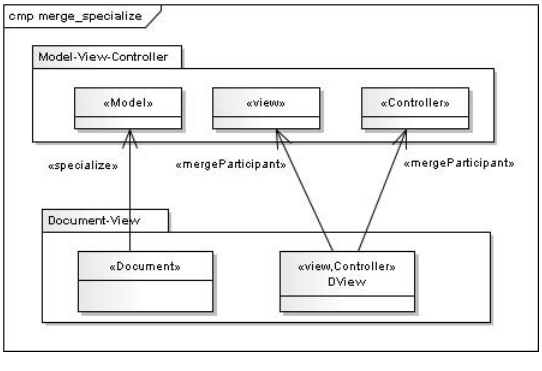

Fig. 2. The mergeParticipant relationship

software architecture). Figure 2 shows an example of the occurrence of *mergeParticipant* relationship among the participants of architectural patterns.

employ: In the employ relationship, participants of a pattern make use of another pattern for their complete implementation. Patterns using the 'employ' relationship are often used together within a software architecture where one pattern often 'makes use of' another pattern to fulfill specific design needs. Patterns having an 'employ' relationship can be applied separately to a software architecture as the relationship does not constrain the presence of both patterns within the architecture. For instance, the MVC pattern often employs the Observer pattern for implementing the change propagation mechanism. However, each of these patterns can also be individually applied to a software architecture. Figure 3 shows the employ relationship among the participants of the Iterator and Batch Method patterns.

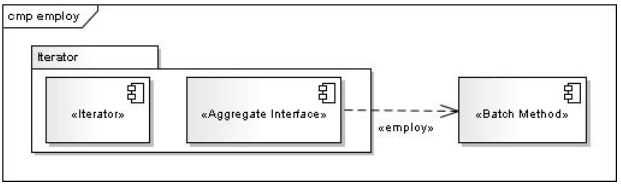

Fig. 3. Employ relationship between the Iterator and Batch Method Patterns

depends: The depends relationship shows the need of pattern participant(s) to use another pattern for their complete implementation. In comparison to the employ relationship, the depends relationship is a strong dependency of a pattern's participants on another pattern where participants of the source pattern are seldom applied without the use of target pattern participants. The depends relationship is shown in a particular example of Client-Server and Broker pattern in figure 4.

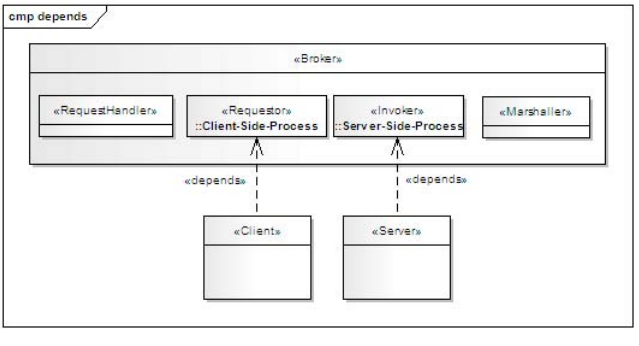

Fig. 4. The depends relationship among pattern participants

importPattern: In the importPattern relationship, the participants of the target pattern import all participants from the source pattern i.e. all participants of a pattern are

modeled within the participant of another pattern. The importPattern relationship is similar to Package import in UML, Family import in ACME, etc. For instance, individual layers in the Layers pattern can import other patterns e.g. the Pipes and Filters pattern is used for implementing data processing layers as illustrated, for example, in figure 5.

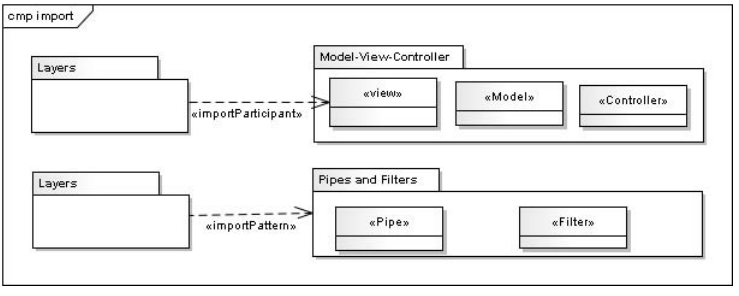

Fig. 5. importPattern and importParticipant relationships among pattern participants

importParticipant: In the importParticipant relationship, participants of the target pattern import 'specific' participants from the source pattern. For instance, an individual layer in the Layers pattern can import the View and Controller participants of the MVC pattern while the Model participant of the MVC pattern resides in another layer. Figure 5 shows an example of using *importParticipant* relationship.

interact: In the interact relationship, certain participants of the source pattern 'interact' with the participants of the target pattern. This relationship represents a loose coupling among the participants of different architectural patterns such as events, procedure calls, etc. For instance, the Request Handler pattern has an interact relationship with the Proxy pattern to send and receive messages as shown in figure 6.

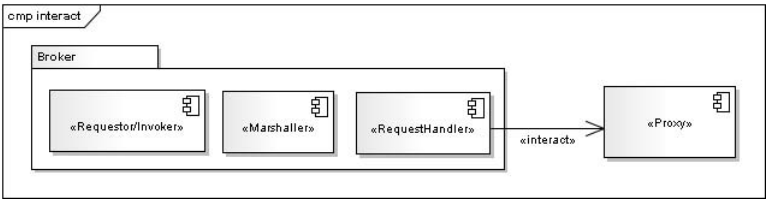

Fig. 6. The interact relationship

Table 1 provides a number of examples of pattern participants relationships discovered in pattern combinations in the studied architectures.

Table 1. Pattern-Participants Relationships Description

Pattern A	Pattern B	Relationship	Description
Reactor	Leader/ Follower	absorbParticipant	EventHandler and Handle participants are present in both patterns
Reactor	Acceptor-Connector	absorbParticipant	EventHandler and Handle participants are present in both patterns
Layers	Pipes and Filters	importPattern	A specific layer can internally implement Pipes and Filters structure
Layers	MVC	importParticipant	Individual layers can import specific participants of MVC pattern e.g. Model may reside in one layer while View and Controller reside in another layer
Layers	Client-Server	importParticipant	see previous line comments
Layers	Broker	importPattern	A specific layer can be implemented as a Broker
Layers	Proxy	importPattern	A specific layer can be implemented as a proxy to other layers
Layers	Factory Method	importPattern	A layer can be implemented using factory method to handle different requests
MVC	Observer	employ	MVC employs observer pattern to implement change notification mechanism
MVC	Factory Method	importPattern	The Model participant can be implemented using Factory method
Broker	Client-Server	depends	Broker is often modeled in combination with the Client-Server pattern
Client-Server	Proxy	interact	The interaction mechanism between Client and Server may use proxy
Active Object	Proxy	integrate	A proxy can act as an active object
Scheduler	Proxy	interact	A scheduler can monitor the requests to decide when a request needs to be executed

4 Related Work

Several pattern languages have been documented in the literature e.g pattern languages for solving specific design problems [2], domain-specific pattern languages [1], and the pattern languages documented in the Pattern Oriented Software Architecture book series [4]. Buschmann et. al. [1] present a pattern language for distributed computing that includes 114 patterns grouped into 13 problem areas. The problem areas address technical topics related to building distributed applications e.g. Event Demultiplexing, Concurrency, Synchronization etc. Their pattern language serve as an overview about the selection and use of related architectural patterns to solve design problems in specific areas. However, the language in itself presents architectural patterns as components, objects and entities linked through generic textual relationships. For instance Model-View-Controller has a 'request handling' relationship with the Command, Command Processor, Application Controller, and Chain of Responsibility patterns. Our work significantly differs from their work as we document relationships among the 'participants' of architectural patterns that can be used more effectively for combing any two architectural patterns in several different forms.

Some work has been done on proposing patterns languages that address specific architectural concerns such as pattern languages for usability [6], pattern languages for concurrency issues [1], pattern languages for performance-critical systems [2] etc. However, these languages provide relationships that best fit to address the concerns they relate to and do not address the relationships among participants of related architectural patterns. In terms of granularity, pattern languages that deal with specific concerns provide more enriched relationships as compared to general pattern languages but they too do not address the relationships among participants of architectural patterns and

overlook possible variation in relationships among the participants of combined architectural patterns.

In our previous work [3], we have documented relationships among architectural patterns in different architectural views that show specific aspects of systems like data flow, interaction decoupling etc. However, though such relationships provide valuable information about pattern-to-pattern relationships (e.g. communication between Layers may use Pipes and Filters), this language too does not focus on relationships among participants of architectural patterns.

5 Conclusion and Future Work

The novelty of our work lies in discovering relationships among the 'participants' of architectural patterns which has not been fully addressed before. The use of pattern participants relationships for integrating architectural patterns offers an effective way to integrate architectural patterns within software architecture design. In particular, this approach offers: a) reusability by providing a vocabulary of pattern-to-pattern relationships that help combine the participants of selected architectural patterns; b) model validation support by ensuring that the patterns are correctly combined within a software architecture; and c) explicit representation of 'integrated' architectural patterns participants within software architectures.

As future work, we plan to apply our approach to industrial case studies for designing software architectures and by conducting controlled experiments. We are in the process of developing a pattern modeling tool called Primus [7], which will support integrating architectural patterns and pattern variants, modeling pattern variability, architectural views synchronization, and source code generation. We believe that we can discover more pattern participants relationships in the near future, which will provide a better reusability support to software architects for effectively integrating architectural patterns.

References

1. Schmidt, D.C., Buschmann, F., Henney, K.: Pattern-Oriented Software Architecture: On Patterns and Pattern Languages. Wiley Series in Software Design Patterns (2007)
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, vol. 1. Wiley & Sons, Chichester (1996)
3. Avgeriou, P., Zdun, U.: Architectural patterns revisited - a pattern language. Technical Report (2005)
4. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Patterns for Concurrent and Distributed Objects. In: Pattern-Oriented Software Architecture, J. Wiley and Sons Ltd., Chichester (2000)
5. Boosch, G.: Handbook of software architecture: Gallery (2010), `http://www.booch.com/architecture/architecture.jsp?part=Gallery`
6. Patterns and pattern languages of program (2010), `http://hillside.net`
7. Kirtley, N., Kamal, A.W., Avgeriou, P.: Developing a modeling tool using eclipse. In: International Workshop on Advanced Software Development Tools and Techniques, Co-located with ECOOP 2008 (2008)